GPUにおける多倍長演算アルゴリズムとその高速化に関する研究

著者	千葉 光剛
出版者	法政大学大学院理工学研究科
雑誌名	法政大学大学院紀要．理工学・工学研究科編
巻	59
ページ	1-7
発行年	2018-03-31
URL	http://doi.org/10.15002/00021593

GPU における多倍長演算アルゴリズムとその高速化に関する研究

ACCELERATION OF MULTI-PRECISION ARITHMETIC ALGORITHUM ON GPU

千葉 光剛
Mitsutaka CHIBA
指導教員 和田 幸一

法政大学大学院理工学研究科応用情報工学専攻修士課程

GPU is one of the parts in charge of image processing, and has a high parallel computing power when compared to need blank after CPU.GPGPU technique using a high computational power of GPU to general purpose computing have been studied.
In this paper, we consider to accelerate operations on multi-precision integers used when dealing with large values exceeding 32 bits or 64 bits. We implemented a carry look-ahead addition algorithm, a carry save addition algorithm, and a redundant binary addition algorithm on the GPU. The implemented algorithm compares the MPIR of the multi-precision arithmetic library with the calculation time.
As a result, multiple length addition was faster than MPIR under limited conditions, and with multiple length multiplication it was two to three times slower than MPIR.

***Key Words** : GPGPU, Carry look-ahead adder, Carry save adder, Redundant binary representation*

1. はじめに

GPU は画像処理を担当する主要な部品の一つである.GPU は CPU に比べて高い並列処理能力を持っているため,現在 GPU を用いた計算処理の高速化が検討されている[1].GPU を 3D グラフィックス処理以外の汎用計算を行わせる技術が「GPGPU (General Purpose Computation on Graphics Processing Unit)」である.

多倍長整数は,32bitや 64bitを超えるような大きな値のことである.通常 C 言語などで用いられる値は 64bitで表現できる値しか扱うことができない.しかし円周率 π や $\sqrt{2}$などの数学定数の研究や暗号などでは数百万桁のような値を扱う必要があり多倍長整数が用いられることがある.多倍長整数はメモリ容量を超えない範囲で任意に精度を決定することができる.これにより多倍長演算には精度の高いデータの演算を行えることの他にも,オーバーフローを考慮せずにプログラムを書けるという利点があるが,その一方で多倍長演算は処理が重いという欠点もある.そのため多倍長演算を高速化できれば,多倍長演算を用いる精度の高いデータを扱う問題全体の計算時間の高速化にもつながり,多倍長演算の高速化は重要である.[2][3][4][5]

本論文では,n 桁の加算を回路として実現したときに並列化できる桁上先見加算アルゴリズム,桁上保存加算アルゴリズム,冗長 2 進表現を利用した加算アルゴリズムを GPU 上で実装した.桁上先見加算アルゴリズムは回路サイズ O(n)回路段数 O($logn$),桁上保存加算アルゴリズムは回路サイズ O(n)回路段数 O(1),冗長 2 進表現を用いた加算アルゴリズムは回路サイズ(n),回路段数(1)であるので GPU を用いて並列に計算することで多倍長演算の多倍長加算と多倍長乗算の高速化を図った.

多倍長整数は配列を用いて表現しており,配列の 1 つの要素を 1bit とする場合と配列の 1 つの要素を 32bit とする場合を実装した.つまり$nbit$の多倍長整数は 1 要素 $1bit$の場合は要素数n個の配列で表現されており,1 要素 32bit の場合は要素数$n/32$個の配列で表現されている.

加算アルゴリズムとして,桁上先見加算アルゴリズム, 桁上げ保存加算アルゴリズムと冗長 2 進表現を利用した加算アルゴリズムを用いた.多倍長乗算アルゴリズムは, 被乗数と乗数から部分積を求め, その部分積の加算を繰り返す計算に 3 つの加算アルゴリズムを利用して積を求める.

多倍長加算を行うアルゴリズムは, 配列の 1 つの要素を 1bit としたときと 32bit としたとき,それぞれで桁上先見加算アルゴリズムと冗長 2 進表現を利用した加算アルゴリズムを実装した.

多倍長乗算アルゴリズムは, 配列の 1 つ要素を 1bit として加算を桁上先見加算アルゴリズム,冗長 2 進表現を利用した加算アルゴリズムの 2 つを用いた多倍長乗算アルゴリズムを実装した.また配列の 1 つ要素を 32bit として計算する桁上先見加算アルゴリズム,桁上保存加算アルゴリズムと冗長 2 進表現を利用した加算アルゴリズムの 3 つを用いていた多倍長乗算アルゴリズムも実装した.

多倍長加算に関しては 4 つのアルゴリズム, 多倍長乗算に関しては 5 つのアルゴリズムの実行時間を計測し, CPU 上で実行され高速であると知られている多倍長演算ライブラリ MPIR[6]の実行時間と比較を行った.

結果として多倍長加算では,配列の 1 要素を 32bit としたときの冗長 2 進表現を利用した多倍長加算アルゴリズムが,答えが冗長 2 進表現で求まる場合 MPIR よりも速かったが 2 進表現で答えを求める場合には勝つことはできなかった. 多倍長乗算では配列の 1 つの要素を 32bit とした桁上保存加算アルゴリズムを利用した多倍長乗算アルゴリズムが MPIR よりも 2 倍遅く,冗長 2 進表現を利用した加算を用いた多倍長乗算アルゴリズムが MPIR より 3 倍遅いことが分かった.

2. GPU

GPU は,グラフィックボードなどに搭載され,3D グラフィックスの生成・表示など画像処理を行うために特化した専用プロセッサである.GPU は同一の構造を持つ多数の要素プロセッサを内蔵しており,それらの要素プロセッサに処理を分散させ,演算を同時実行させる並列処理によって高速化を実現している. この高い並列処理能力を利用して,グラフィックス処理以外の汎用演算などに応用する技術を,GPGPU(General-Purpose computing on GPU)と呼ぶ.[1]

GPGPU の開発環境である「CUDA」は NIVIDIA 社から提供される C 言語を拡張した GPU を用いた汎用計算処理プログラムを開発するための総合開発環境である.

CUDA では,CPU と GPU の両方を用いて計算を行うことができる.GPU で実行されるプログラムはカーネル関数と呼ばれる,カーネル関数内のプログラムはスレッドと呼ばれる実行単位で実行され全てのスレッドで同じプログラムが実行される.スレッドはそれぞれ固有のスレッド ID を持っており,そのスレッド ID を用いて同じプログラムを実行していても異なった配列にアクセスするなどしてそれぞれ並列に固有の処理を実行できる.

3. 多倍長整数

通常 C 言語などで用いられる値は 64bit で表現できる値しか扱うことができないため,64bit で表現できる値を超えた場合オーバーフローを起こしてしまうため,円周率πや√2 などの数学定数の研究や暗号などの数百万桁のような値を扱う問題を演算することはできない.しかし多倍長整数はメモリ容量を超えない範囲で任意に精度を決定することができるので,精度の高いデータの演算を行うことができる.

本論文では, nbit の多倍長整数は 1 要素 1bit の場合は要素数 n 個の配列で表現されており,1 要素 32bit の場合は要素数 n/32 個の配列で表現されている.多倍長整数の実装例を図 1 に示す.

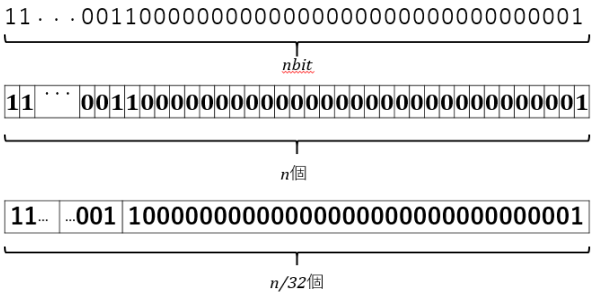

図 1　多倍長整数の実装例

本論文では多倍長演算として多倍長加算と多倍長乗算の実装を行った. 加算アルゴリズムとして *n* 桁の加算を回路として実現したときに並列化できる桁上先見加算アルゴリズム,桁上保存加算アルゴリズム,冗長 2 進表現を利用した加算アルゴリズムを GPU 上で実装した.桁上先見加算アルゴリズムは回路サイズ O(*n*)回路段数,O(*logn*)実行でき,桁上保存加算アルゴリズムは回路サイズ O(*n*), 回路段数 O(1)で実行でき,冗長 2 進表現を用いた加算アルゴリズムは回路サイズ(*n*),回路段数(1)で実行できるので GPU を用いて並列に計算することができる.乗算は被乗数と乗数から部分積を求め,部分積の加算を繰り返すことで積を求めている.部分積の加算に上記の 3 つのアルゴリズムを用いた.乗算の部分積の求め方は,配列の 1 つの要素を 1*bit* としたときは筆算と同じ要領で乗数の 1 がある *bit* に応じてシフトを行って部分積を求めており,32*bit* としたときも筆算と同じ要領で部分積を求めているが被乗数と乗数の要素の積を 64*bit* として求め,32*bit* のズレが

存在するグループと存在しないグループに分けて 2 つの部分積を求めている.

今回比較に用いたライブラリは多倍長演算ライブラリの GMP の互換性ライブラリ,「MPIR[6]」である.MPIR は C,C++で使用可能であり,C++で用いる場合は多倍長数クラスを使用することにより,数値や文字列で多倍長数の入力をすることができる.多倍長演算では四則演算などはサブルーチンを呼び出さなくても実行できる.

4. 桁上先見加算アルゴリズム[7]

桁上先見加算アルゴリズムとは,各桁の桁上げの状態を求め,下位桁の桁上げの状態を,並列プレフィックス和計算を行うことで下位桁からの桁上げを求めるというものであり, *nbit* の加算を回路サイズ O(*n*),回路段数 O(*logn*)で実行できる.桁上先見加算アルゴリズムは各桁の状態を求める KPG アルゴリズムと求めた各桁の状態の並列プレフィックス和計算をおこなうアルゴリズムの 2 つのアルゴリズムから成る.

5. 桁上保存加算アルゴリズム[8]

桁上保存加算アルゴリズムでは,$nbit$の多倍長整数x,y,zの加算を行い,各桁の和$u(nbit)$と桁上げ$v(n+1bit)$を求めることで 3 数の加算を 2 数に変換することができ, *nbit* の加算を回路サイズ O(n),回路段数 O(*logn*)で実行できる.これは $x+y+z$ を $u+v$ で表現しており,和u_iはx_i,y_i,z_i の排他的論理和で求まり, 桁上げv_iはx_{i-1},y_{i-1},z_{i-1} の多数決関数で求めることができる.

6. 冗長 2 進表現を利用した加算アルゴリズム

冗長 2 進表現とは,Avizienis が文献[9]で提案した SD(Singned Digit)表現の一種で基数 2 の拡張 SD 表現である.通常の 2 進表現と同様に,下位からi番目の桁は2^{i-1}の重みをもつが,各桁は{-1,0,1}である.冗長 2 進表現で$x_nx_{n-1}\cdots x_1(x_i\in\{-1,0,1\})$は,$\sum_{i=1}^{n}x_i\cdot 2^{i-1}$という数を表す.以後-1 は$\bar{1}$と表し,冗長 2 進表現の数は$[x_nx_{n-1}\cdots x_1]_{SD2}$, 2 進表現は$[x_nx_{n-1}\cdots x_1]_2$と表記する.冗長 2 進表現では 1 つの数をいくつかの冗長 2 進表現で表すことができる.

符号なし 2 進表現で$[x_nx_{n-1}\cdots x_1]_2$は$\sum_{i=1}^{n}x_i\cdot 2^{i-1}$という数を表しているので,冗長 2 表現で$[x_nx_{n-1}\cdots x_1]_{SD2}$とすることができる.つまり符号なし 2 進表現で表された数は変換することなく冗長 2 進表現で表された数として扱うことができる.冗長 2 進表現から 2 進表現への変換は,2 つの 2 進数の減算により行うことができる.冗長 2 進表現で$[x_nx_{n-1}\cdots x_1]_{SD2}$は,$\sum_{i=1}^{n}x_i\cdot 2^{i-1}=\sum_{x_i=1}2^{i-1}-\sum_{x_i=\bar{1}}2^{i-1}$という数を表しているので 1 の桁だけで作った 2 進数と$\bar{1}$の桁だけで作った 2 進数とで減算を行うことで 2 の補数表示の 2 進表現に変換することができる.従って *n* 桁の冗長 2 進表現された数は,*n*+1*bit* の 2 の補数表示の 2 進表現の数に変換される.

冗長 2 進表現を利用した加算[10]ではその冗長性を利用して,桁上げが高々1 桁しか伝搬しないようにすることができ,*nbit* の加算を回路サイズ O(*n*)回路段数 O(1)で実行することができる.2 進表現では,加算を行う場合 *i* 桁において下位から桁上げがありx_i,y_iのどちらか一方が 1 の場合桁上げが伝搬してしまう.冗長 2 進表現では「1」は 2 桁で$[1\bar{1}]_{SD2}$と$[01]_{SD2}$の 2 通りで表すことができるので,0+1=$[1\bar{1}]_{SD2}$と考え,中間桁上げc_iを 1,中間和s_iを$\bar{1}$として予め上位に桁上げをしておくことで下位桁からの桁上げを吸収することができる.しかし$[c_is_i]$を常に $[1\bar{1}]$ にしてしますと下位から$\bar{1}$が桁上げされてきた場合に $\bar{1}$ が伝搬してしまう.そのため下位からの 1 の桁上げの可能性の有無を判断し,可能性がある場合は$[1\bar{1}]$として,ない場合は[01]としなければならない.また下位からの$\bar{1}$の桁上げがある場合には[01]で伝搬を吸収することができる.同様にx_i,y_iのどちらか一方が$\bar{1}$の場合も「$\bar{1}$」は$[\bar{1}1]_{SD2}$と$[0\bar{1}]_{SD2}$の 2 通りで表すことができるので,下位から 1 の桁上げの可能性がある場合は$[0\bar{1}]$,下位から$\bar{1}$の桁上げの可能性がある場合は$[\bar{1}1]$とすればよい.以上より場合分けをする必要がありそれは下位桁が両方とも非負の場合とどちらか一方が負の場合で分けることで下位桁からの桁上げが吸収され伝搬しないようにすることができる.よって$nbit$の多倍長整数のx,yの加算は,ある桁x_i,y_iとその下位桁x_{i-1},y_{i-1}から中間和s_iと中間桁上げc_iを求め,和$w_i=s_i+c_{i-1}$を計算することで中間和s_iと下位からの桁上げc_{i-1}から和w_iを求めることができる.w_iはs_iとc_{i-1}のみから定まり,桁上げは起こらない.ある桁x_i,y_iとその下位桁x_{i-1},y_{i-1}から中間和s_iと中間桁上げc_iを求めるときの計算規則を表 1 に示す.

表 1 冗長 2 進表現を利用した加算の計算規則

被加数 (x_i)	加数 (y_i)	1 つ下位の桁 (x_{i-1},y_{-1})	桁上げ (c_i)	中間和 (s_i)
1	1	—	1	0
1	0	両方とも正	1	-1
0	1	少なくとも一方負	0	1
0	0	—	0	0
1	-1			
-1	1			
0	-1	両方とも正	0	-1
-1	0	少なくとも一方負	-1	1
-1	-1	—	-1	0

冗長 2 進表現を利用した加算アルゴリズムは配列の 1 つの要素を 1*bit* として実装したアルゴリズム[11]と 32*bit* として実装したアルゴリズムと 2 つのアルゴリズムがある.

配列の 1 つの要素を $1bit$ としたときのアルゴリズムは,配列の要素を$\{\bar{1},0,1\}$とし $nbit$ の多倍長整数を冗長 2 進表現で表し,表 3 をテーブルとして実装して $x_i, y_i, x_{i-1}, y_{i-1}$から$s_i$,$c_i$を決定できるように実装されている[11].
配列の 1 つの要素を $32bit$ としたときには,2 つの配列を用意し,bitごとの値を見ることで$\{\bar{1},0,1\}$の判断を行うことができるようにした.
冗長 2 進表現のエンコードを表 2 に示す.

表 2 冗長 2 進表現のエンコード

x	$x1$	$x0$
$\bar{1}$	1	0
0	0	0
1	0	1

表 2 より 2 つの配列で正の 1 ならば$x0$が 1 となり，負の 1 ならば$x1$が 1 となる.このエンコードをもとに表 1 の計算規則を作り直しbit演算で計算ができるように数式を求めた.作り直した計算規則を表 3 に示す.また和w_iを求めるための表を表 4 に示す.
fは下位桁の 1 の桁上げの可能性があるのか$\bar{1}$の桁上げがあるのかを判断する数式で$x1$ と$y1$の論理和を求めることで,下位桁が両方非負ならば 0,少なくとも一方負ならば 1 となる.

表 3　2 つの配列を用いた場合の冗長 2 進表現を利用した加算の計算規則

$x1_i x0_i$	$y1_i y0_i$	$x1_{i-1}, y1_{i-1}$	$c1_i c0_i$	$s1_i s0_i$
01	01	$f = x1_{i-1} + y1_{i-1}$	01	00
00	00		00	00
01	10		00	00
10	01		00	00
10	10		10	00
01	00	両方非負	01	10
00	01	$f = 0$	01	10
10	00		00	10
00	10		00	10
01	00	少なくとも一方負	00	01
00	01	$f = 1$	00	01
10	00		10	01
00	10		10	01

表 3 より$c1, c0, s1, s0$を求める数式を以下に示す.

$$c1_i = x1_i \cdot \overline{x0_i} \cdot y1_i \cdot \overline{y0_i} + ((x1_i \oplus y1_i) \cdot \overline{x0_i + y0_i}) \cdot f$$

$$c0_i = \overline{x1_i} \cdot x0_i \cdot \overline{y1_i} \cdot y0_i + (\overline{x1_i + y1_i} \cdot (x0_i \oplus y0_i)) \cdot \bar{f}$$

$$s0_i = (x1_i \oplus x0_i \oplus y1_i \oplus y0_i) \cdot f$$

$$s1_i = (x1_i \oplus x0_i \oplus y1_i \oplus y0_i) \cdot \bar{f}$$

⊕:排他的論理和

表 4 $c1_{i-1}c0_{i-1}$と$s1_i s0_i$の加算

$c1_i c0_i$	$s1_i s0_i$	$w1_i w0_i$
00	00	00
10	01	00
01	10	00
00	01	01
01	00	01
10	00	10
00	10	10

表 4 より$w1_i w0_i$を求める式を以下に示す.

$$w1_i = \overline{(c0_i + s0_i)} \cdot (c1_i \oplus s1_i)$$
$$w0_i = \overline{(c1_i + s1_i)} \cdot (c0_i \oplus s0_i)$$

7.　実験

実験環境を表 5 に,GPU のスペックを表 6 に示す.

表 5　実験環境

CPU	Intel Core i7-3090X
GPU	GeForce GTX TITAN
MPIR のバージョン	3.0.0

表 6　GeForce GTX TITAN

GPU	GTX TITAN
演算コア数	2688 基
搭載メモリ容量	6GB
演算ユニット	14 基
ワープごとのスレッド数	32
ブロックごとの最大スレッド数	1024

GPUに実装したアルゴリズムの実行時間はCPU-GPU間の通信時間を含まない GPU 上でアルゴリズムが実行されている時間である.また冗長 2 進表現を利用した加算,乗算は冗長 2 進表現で答えが求まるまでの時間であり,桁上保存加算アルゴリズムを利用した加算は 3 数の加算を行い 2 数が求まる時間,乗算は部分積の 32 bitのズレがあるグループと無いグループで 2 数ずつ計 4 つが求まるまでの時間であり,桁上先見加算アルゴリズムは加算,乗算ともに答えが求まる時間である.

多倍長加算アルゴリズム 4 つと MPIR との実行時間の比較を表 7,図 2 に示し,冗長 2 進表現を利用した 2 つの多倍長加算アルゴリズムの実行時間の比較を図 3 に示す. また配列の 1 つの要素を 32bit とした桁上保存加算アルゴリズムは 3 数の加算を 2 数に変換するため加算として比較することができないが,乗算アルゴリズムでは部分積の加算の繰り返しになるため部分積の 1 回の加算にかかる実行時間を知るために計測を行い,桁上保存加算アルゴリズムと冗長 2 進表現を利用した加算の実行時間の比較を表 8 に,冗長 2 進表現を利用した 2 つのアルゴリズムで配列の要素数を同じにしたときの実行時間の比較を表 9 に示す.

多倍長乗算アルゴリズム 5 つと MPIR との実行時間の比較を表 10,図 4 に示し,配列の 1 つの要素を 32*bit* としたときの桁上保存加算アルゴリズムを用いた多倍長乗算アルゴリズムと冗長 2 進表現を利用した加算アルゴリズムを用いた多倍長乗算アルゴリズムと MPIR との実行時間の比較を図 5 に示す.

表 7　多倍長加算の実行時間の比較(単位は ms)

bit	先見加算 (1bit)	先見加算 (32bit)	冗長 2 進 (1bit)	冗長 2 進 (32bit)	MPIR
262144	3.742	0.469	0.182	0.105	0.015
524288	7.206	0.813	0.218	0.112	0.030
1048576	15.06	1.509	0.343	0.129	0.060
2097152	31.35	3.217	0.461	0.136	0.117
4194304	64.38	7.543	0.768	0.178	0.234

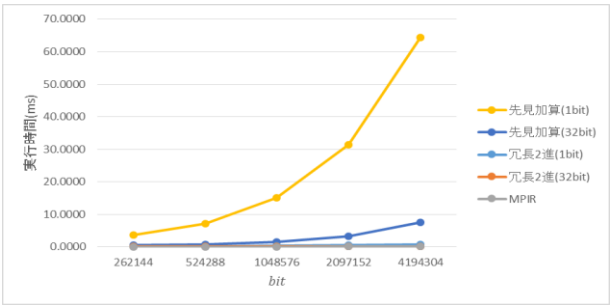

図 2 4 つの多倍長加算アルゴリズムと MPIR との実行時間の比較

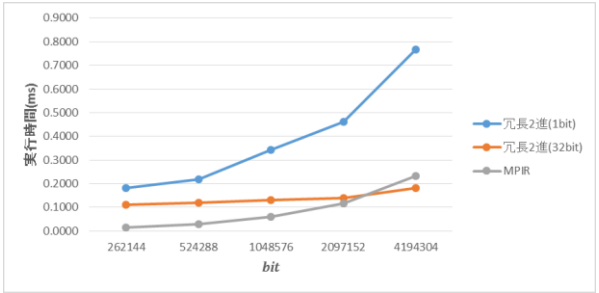

図 3 冗長 2 進表現を利用した多倍長加算アルゴリズムと MPIR との実行時間

表 8　32*bit* とした桁上保存加算と冗長 2 進表現を利用した加算の比較(単位は ms)

bit	桁上保存加算(32*bit*)	冗長 2 進(32*bit*)
16320	0.0836	0.1095
32640	0.0898	0.1108

表 9　冗長 2 進表現を利用したアルゴリズムの比較(単位は ms)

要素数	冗長 2 進 (1*bit*)	冗長 2 進(32*bit*)
262144	0.1822	0.2389
524288	0.2184	0.3454
1048576	0.3430	0.5962
2097152	0.4614	0.9321
4194304	0.7678	1.7580

表 10　多倍長乗算の実行時間の比較(単位は ms)

bit	桁上先見 (1*bit*)	桁上先見 (32*bit*)	桁上保存 (32*bit*)	冗長 2 進 (1*bit*)	冗長 2 進 (32*bit*)	MPIR
2048	120.5	1.03	0.25	5.20	0.47	0.01
4096	479.7	2.53	0.31	20.16	0.54	0.04
8192	1899.7	8.91	0.64	83.81	0.70	0.11
16384		36.4	1.01	320.72	1.02	0.26
32768		151.0	1.45		2.03	0.71

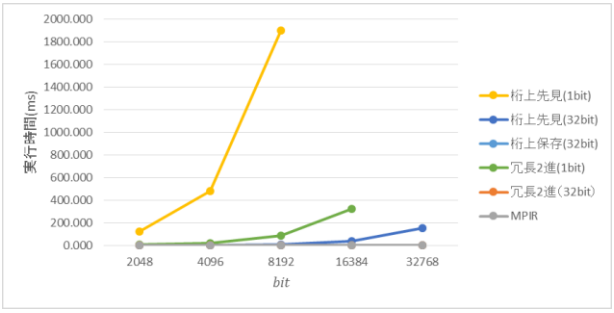

図 4 5 つの多倍長乗算アルゴリズムと MPIR との実行時間の比較

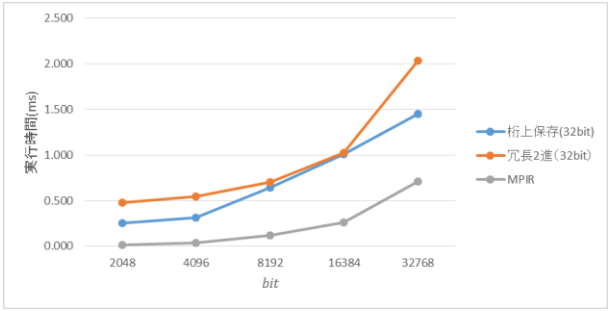

図 5 2 つの多倍長乗算アルゴリズムと MPIR との実行時間の比較

表7,図2,4より,実装された多倍長加算アルゴリズムで配列の要素を32bitとしたときの冗長2進表現を利用した加算アルゴリズムがMPIRよりも高速であることが分かる.しかし答えは冗長2進表現で求まっているため2進表現に変換する場合には2進表現へ変換を行う必要があり,変換は加算を用いて行われるため2進表現での加算の答えを求めたい場合にはMPIRの方が速いと考えられる.冗長2進表現を利用した多倍長加算アルゴリズムがMPIRより高速である理由は,GPU上で並列にbit演算で計算できるためCPU上で逐次計算しているMPIRよりも高速になったのだと考えられる.また桁上先見加算アルゴリズムを用いた2つの多倍長加算アルゴリズムはMPIRよりも遅いことが分かる.より速かった配列の1つの要素を32*bit*とした桁上先見加算アルゴリズムを利用した多倍長加算アルゴリズムがMPIRよりも遅い理由としてbit演算でアルゴリズムを実行しなければならないためアルゴリズムの過程でシフト演算を多用して必要演算を行っているため演算が多くなってしまい実行時間がかかってしまったと考えられる.

表8より,多倍長乗算アルゴリズムで部分積の加算を行う際に用いる多倍長アルゴリズムとして桁上保存加算アルゴリズムは冗長2進表現を利用した加算アルゴリズムよりも高速であることがわかる.これは桁上保存加算アルゴリズムの方が冗長2進表現を利用したアルゴリズムよりも*bit*演算が少ないからであると考えられる.

表9より,要素数を同じにした場合配列の1つの要素を1*bit*とした冗長2新表現を利用した加算アルゴリズムの方が高速であることが分かる.これは扱っている*bit*数の違いが32*bit*あるからであるが,アルゴリズムとしては32*bit*を計算するアルゴリズムは演算が多いため1bitを計算するアルゴリズムよりも遅くなったと考えられる.

表10,図4,5より,今回実装したアルゴリズムではMPIRより遅いことが分かる.実行時間が近い2つのアルゴリズムに関して桁上保存加算アルゴリズムでは32*bit*シフトと2回の桁上保存加算アルゴリズムと2数の加算を行う操作が残っているが32*bit*シフトは0.06181ms,2回の桁上保存加算アルゴリズムは0.1672ms,2数の加算はMPIRで0.005639msで計算できているので全てを足すと1.6882msとなり,MPIRよりも2倍遅いと考えられる.冗長2進表現を利用した加算では冗長2進表現の積を2進表現に変換しなければならないが,変換に必要な計算は$\bar{1}$が格納された配列を反転させて1が格納された配列と加算を行うことで2進表現に変換することができる.反転の計算は0.007012msで行うことができ,加算はMPIRでは0.005639msで行うことができているので全てを足すと2.0495msとなりMPIRよりも3倍遅いと考えられる.桁上先見加算アルゴリズムを用いた多倍長乗算が遅い理由は多倍長加算アルゴリズムが遅い理由と同じであり,多倍長乗算は部分積の加算を繰り返すことで実現しているので多倍長加算アルゴリズムが遅ければ遅いほど乗算アルゴリズムも遅くなってしまうからだと考えられる.桁上保存加算アルゴリズムが遅い理由は,和と桁上げなどの並列できる計算を並列に行っていないからであると考えられ,マルチスレッドアルゴリズムを用いることで並列に計算できると考えられる.冗長2進表現を利用した加算が遅い理由は,下位からの桁上げの有無を表すfと中間桁上げで1bitシフトを行わなければならないため要素数が増えると桁上保存加算に比べて計算時間が遅くなっていると考えられる.また冗長2進表現を利用した加算アルゴリズムを用いた多倍長乗算アルゴリズムでも並列に計算できる数式を含んでいるのでマルチスレッドアルゴリズムを用いることで並列に計算することができると考えられる.

8. まとめ・今後の課題

多倍長加算アルゴリズムに関しては冗長2進表現での答えであれば冗長2進表現を用いた加算アルゴリズムが高速であるが,2進表現で答えを求める場合にはMPIRに実装したアルゴリズム4つは勝つことができなかった.桁上先見加算アルゴリズムに関してはシェアードメモリを用いることで高速化を図ることができるが演算の多さの観点からMPIRの実行時間に勝つことは難しいと考えられる.

多倍長乗算アルゴリズムに関しては桁上保存加算アルゴリズムと冗長2進表現を利用した加算アルゴリズムの2つがMPIRよりも高速化であると考えられ,桁上保存加算アルゴリズムがより高速であると考えられる.桁上保存加算アルゴリズムを用いた多倍長乗算の課題としては,シェアードメモリを用いた計算の高速化を図ること,マルチスレッドアルゴリズムを用いた計算の並列化を図ることの2つが今後の課題である.冗長2進表現を利用した加算アルゴリズムの課題は,シェアードメモリを用いた計算の高速化,マルチスレッドアルゴリズムを用いた計算の並列化を図ることと今回エンコードとして用いたものよりも優れたエンコードはないか調べることの3つが今後の課題である.

参考文献

[1] 青木尊之,額田彰.”はじめてのCUDAプログラミング”. 光学社(2009)

[2]藤田峻太,藤本典幸,澤田幸一郎,紫垣賢人,川口大輔.”多倍長演算ライブラリ MPIR互換のCUDAライブラリの実装について”. 第13回 情報科学ワークショップ (2017)

[3] 澤田幸一郎,藤本典幸,和田幸一.”CUDAを用いた多倍長乗算の実装について”. 第13回 情報科学ワークショップ (2017)

[4]藤本典幸,紫垣賢人.”CUDAを用いた多倍長加減算の実装について”. 第13回 情報科学ワークショップ (2017)

[5]川口大輔,藤田峻太,藤本典幸. ”CUDA を用いた多倍長除算と多倍長平方根演算の実装について”. 第 13 回 情報科学ワークショップ (2017)
[6] T. Granlund, the GMP Development Team, W. Hart,and the MPIR Team. MPIR 2.7.2
"*MPIR* 2.7.2" , http://mpir.org/downloads.html. (November 2015).
[7] 武居知哉. ” GPU を用いた桁上先見加算アルゴリズムを使用する多倍長演算の高速化について”. 法政大学理工学部応用情報工学科卒業論文(2018-2)
[8] 尾操一. “GPU を用いた桁上げ保存加算を利用した多倍長乗算の高速化について”. 法政大学理工学部応用情報工学科卒業論文(2018-2)
[9]Avizieis ,A. ”Signed Digit Number Representations for Fast Parallel Arithmetic”, IRE Trans. Electronic Comput., Ec-10, 9, pp.389-400(September 1961)
[10] 高直史,安浦寛人,矢島脩三”冗長 2 進加算木を用いた VLSI 向き高速乗算器”,電気通信学会論文誌 J67-D(4), (1984)
[11] 高信裕.”GPU を用いた冗長 2 進表現を利用した多倍長整数乗算の高速化について”, 法政大学理工学部応用情報工学科卒業論文(2018-2)

発表学会

千葉,和,藤本,尾崎,高木,武居.”GPU における多倍長演算とその高速化に関する研究” 2018 年電子情報通信学会 ISS 特別企画 学生ポスターセッション(2018-3-予定)